Holographic Learning

With fig, computers and coding

First Draft Version, Sep 2018

Preface

Holographic learning is an effort to make education more like the natural process of learning. The purpose is to make learning easier, more efficient, and more enjoyable.

Holographic learning is an alternative to having everything first broken into brick-by-brick pieces, and then handed one at a time, in some specific order, to the learner to try to put together.

Inspired in part by Buckminster Fuller, the idea is to-- instead of digesting the world and then regurgitating it into the mouths of recipients like baby birds-- take the learner on a tour of the entire subject and point out the sights. Rather than trying to build the whole thing from puzzle pieces, we will try to take the whole thing apart, focusing on the easier-to-understand parts.

The step after holographic learning is refinement. Introduction leads to familiarity, which leads to further details. This is natural, too-- learning is supplemented by experience in daily life. But rather than treat school as a separate context, where someone graduates from school to something real-- we will start from the other end and work back to details.

The reason we call this "holographic learning" is that you are encouraged to spend less effort on a particular point in a chain, where any particular point may waste your time on something relatively unimportant, and to just *keep going.*

Natural learning isn't one-dimensional-- if you learn a, c, d through z, you do not have go back and learn the alphabet over-- you simply insert "b" in the right place and keep moving forward. So a lesson plan that makes you spend way too much time where you're stuck is incredibly inefficient.

Of course if you want to stop somewhere to focus, stop and focus-- smell a rose. But don't let your education be a cattle chute. When the instructor goes home, learning goes on, and if you don't know the "proper order" of items to learn, it wont stop you from refining the experience on your own time. So why should it, when your education is "on the clock?"

"If the mat was not straight, the master would not sit."

-- the unfortunate foundation of our global, Confucian educational system

The role of teacher is a servant's role. The teacher assists the learner in their own, personal role of learning. A true master of teaching is the one who assists the learner in reaching their own level of mastery. A teacher may "know better," but their job is not to know-- their job is to help someone else know.

If the goal of teaching is the learner's self-mastery, then the goal of pedagogy is the teacher's self-mastery-- to provide tools, most often in the form of concepts, that assist the teacher in assisting the learner.

In its purest form, education means that everyone is learning. An overly rigid system of teaching would mean that education stops and waits for itself to happen; if the learner does not have a "b" then they will never obtain the rest of the alphabet.

If for whatever reason, "b" is holding you back-- forget about it, and move to c. You may discover it later, it may become reasonable all of a sudden, and too much of learning is waiting for the next step when the mind naturally leaps-- often correctly.

As much as the teacher's role is to assist the learner, it is also the role of the school to assist the teacher. While some teachers do triumph against the odds, the critiques in this book are aimed at traditional education itself, not the instructors. What we really want to change is the school itself.

Chapter 1: The Holographic Foundation

Like other examples of education, holographic learning has subjects. The difference is in the rigidity of the subject.

Natural learning is less about memorising discrete fact after discrete fact. Even machine learning is less about this--computers are far better at retrieving discrete facts than people are, but doing so doesn't make a computer "smart" at all.

A lot of natural learning is as much about the connections between ideas, rather than just the ideas. So if we want a more natural system of learning, we want to make it easier to connect ideas together.

The rote system of brick-by-brick learning is like a one-dimensional chain of concepts to say "we learned." The ideas themselves have connections, but we focus too much on the chain from the previous idea to the next.

We deny ourselves most of the learning process, unless it happens accidentally. When it does, we generally move "forward" instead of staying and learning more. After all, its a

long chain ahead and we only have so many days in the year.

Like with traditional education, the first job of holographic learning is to lay a foundation. But holographic learning in any subject has two foundations, and three dimensions:

Foundations:

1. The sum of the learners experience so far
2. The overview of the subject at hand

Dimensions:

1. Building concepts onto the first foundation
2. Building concepts onto the second foundation
3. The natural spirit of exploration

The 1st dimension: rather than just accept these ideas, try to look back on your experiences with education so far-- How is this different? How is it the same? Could it improve anything for you in some way?

The 2nd dimension: right now, you are getting an overview of the learning concept itself. Soon, this will move towards the subject being learned. Does knowing something about the approach to teaching help? Might this approach also inspire something more directly relevant to computing?

The 3rd dimension: perpendicular to the 1st dimension, which compares present (new) ideas with past experience-- what additional ("future") ideas are inspired by the subject matter?

In this approach to learning, you are encouraged to think about all of these angles-- but you do not have to always think about each one equally, or even consciously. Sometimes it will be useful to ask questions like these.

Ultimately, holographic learning perpetuates education by turning the quest for knowledge into further quests. By "quests" we mean this very similarly to questions. Science is less about answers, and more about moving towards better questions.

Technically, you don't need a book to learn how to code. There are plenty of tutorials online, you can just look up "Learn how to code." But if you do, you will need to explore many options that might take a very long time to learn-- or, they might require you to start a longer path that's harder to follow.

Should we start out easy? We should, if we want the largest number of people to be able to start with us.

Here is where we change the primary subject to computers.

If you were interested in teaching, we are still really talking about that as well-- but our focus has shifted (as promised) from teaching to computers, and we are demonstrating one concept using the other.

One of the nice things about computers as a subject, is that every fundamental thing a computer does can be described with code-- code exists to run the computer, and the computer exists to run code.

Nicer than that, is that we have a simple language to make all of this easier. In fact, computer languages are themselves holographic (in the sense of our learning paradigm) because when you learn one computer language, you are generally learning the concepts found in countless others.

You are also learning about the fundamentals of computing. This is not just basic trivia, or the history of computing or "behind the scenes." to understanding code is to understand the computer-- if you can make coding easy enough, then everyone can understand computers as well. That's a very powerful bridge.

In the 1980s, people were on their way to understanding computers, but we changed something in the 90s-- we switched from teaching computing to application training.

Many people (probably most people) in the 1980s were computer illiterate, and not at all ashamed to say so-- but we moved from a computer illiterate population that had education as an option for practically everybody, to a computer illiterate population that uses computers and is less likely to ever learn about them in a way that they really understand them.

So we've lost a major opportunity for education, because as we said in the 1980s, "computers are everywhere"-- now they're in your pocket and paying for restaurant meals like a wallet. And holding your entire photo album. And connected to your cell phone.

And so many other things-- the car metaphor doesn't work anymore-- if your entire office at work was bugged and had countless things in it that you didn't know what they were, you would probably want to know more about your office at work. But computer education leaves most people relatively helpless, trying to learn applications instead of computing.

In "Everything I Know," Buckminster Fuller sets out to understand the universe by defining the word "universe" and then creating a fundamental, universal and natural "structure."

The fundamental element of computers on the physical level, is on/off switches and the binary number system that we can use with them. Conceptually however, it is suitable to say that the fundamental element of computers is actually numbers.

No matter what task you're thinking of a computer doing, it is all being done with numbers (and actually whole numbers. And in fact, only 0 and 1-- but that's not how we conceptualise computing in general. 0-10 is fine.)

Chapter 2: The holographic subject of computing

Any subject can be treated holographically, though computing lends itself to it particularly because everything they do is so closely related with numbers and layers of representation.

That doesn't mean this book will delve deeply into math; you really don't have to be great at math to become good with computers. Instead, computers will make math easier to learn-- even if you know very little about it.

Computers can be used to help learn practically anything, depending on the context, but for math in particular the way computers can help most is by demonstrating math and bringing it into greater relevance.

The computer can do math for you, it can be told to do math for you and by you, and it can show you math in different ways, one of which could be more intuitive to you than another.

Most of the time, instead of math, our treatment of computing will deal more with numbers than operations on numbers. The relationships between these numbers can often be

found through addition, subtraction, division and multiplication, but that won't be our focus-- our focus will be representation.

The term "computer" originally applied to a person, whose job it was to perform calculations. The calculator would be given a list of calculations to perform, and he or she would find the answers by doing the math on paper.

The earlier use of the term "pocket computer" was for tables of data, often printed on paper or plastic-- and manipulated by the user who would move one or more little windows cut into one piece of the "computer" over the tables.

When the desired input was in one window, the desired output was available in the right. Many pocket computers of this type are trivial to implement with a modern computer program-- one example being a calculator that converts from Celsius to Fahrenheit or the other way around. Imagine these two lists arranged side-by-side on a circular paper dial:

c

```
21.5| 22 |22.5| 23 |23.5| 24 |24.5| 25 |25.5
    |    |    |    |    |    |    |    |
70.7|71.6|72.5|73.4|74.3|75.2|76.1| 77 |77.9
```

f

If you were moving the window over the Celsius row to find 23.5, the window for the row below it would reveal 74.3. if you were moving the window over the Fahrenheit row to find 77, the other window would reveal 25 degrees c.

This table is based on the formula **f = c times 9, divided by 5, plus 32**, and was produced with the following program code:

```
def to_f(c):
    return float(c) * 9 / 5 + 32
```

That code is written in Python, a programming language that is popular for both educational and professional use.

In fig, another programming language, the function could be defined like this:

```
function to_f c
now = c times 9 divby 5 plus 32 ; return now
fig
```

This isn't necessarily not the easiest way to solve this problem using either of these languages. There are probably ways to make this code easier, while this way requires less typing.

The paper pocket computer generally dealt with pre-calculated values, while the programming language equivalent deals with actual calculations because it can. A computer program can also work with a table of values like the paper version does; the advantage of calculation is that it can offer greater precision and range than a list on paper.

For the moment, we want to treat computing as broadly as possible. We do want to get to the idea that computing is more or less the same thing as code-- and that by truly understanding one, you understand the other.

With holographic learning, which is heavily inspired by Buckminster Fuller, the goal is to start with the broadest picture possible and then to assist the learner in splitting the picture into parts to examine the relationships between them.

So when you find these facts, hold onto the ones you find interesting and then compare them to each other and ask yourself all the questions you can-- then look for answers to those questions whenever possible. It is those questions and curiosity that will guide your learning.

To assist you in splitting the entire subject into parts to examine them, we will suggest some of the ways the subject can be divided, and take you on a tour of some of the ideas that result.

One way the matter can be divided is chronologically-- from past machines to present. We have started that journey by comparing paper dials to modern programming, and insist that for all the things that have changed, there is a great similarity running throughout the history of computing.

The common thread is dealing with numeric values, which we will get to in more detail. There is also useful data in differences between older and newer computers, however small they arguably are.

Part of holographic learning is what we do for the learner, and another part is how the learner guides their own education. But since we want to show new (or at least, less conventional) ways to approach the subject, it is useful to demonstrate a process so that the learner can find a similar means that is natural to them.

We continue the tour of all computing by separating the machine and the function. We started out by saying they are so closely related as to be almost interchangeable. Nowhere is this more true than in software implementations of computers themselves.

You can write, as a software application, a computer simulation of a computer (called a "virtual computer") and then run the same operating system software running on the computer itself-- inside the program.

This is not just theoretical; suppose you remove all software (including windows) from the drive of a PC. Now, you boot another operating system, using some variety of GNU/Linux.

GNU is an operating system that began development in 1984, and Linux is an operating system kernel that began development in 1991, and together they provide an operating system environment that the user has relatively great control over.

Now on this GNU/Linux system, run a program called Qemu. In most instances, Qemu will run in a window on your desktop, leaving little doubt as to whether it is an application. If you have enough resources (particularly ram) free, you can run another operating system in that Qemu window.

In the GNU/Linux world, there are many varieties of the operating system (comparable to the make and model of a car) and to follow the car metaphor, trying out Puppy Linux (one make and model) in Qemu, while using Refracta (another brand) is a bit like test driving a Dodge Neon from the front seat of a Toyota Camry.

So while using one computer, it is often possible to run at least one (sometimes more) virtual computers and run more than operating system at once, from the same desktop at the same time. This should likely prove that computing and code (as the virtual computer can be implemented entirely as software) are interchangeable concepts.

Alternatively, since the 1980s or prior it was possible to purchase a computer that had some of its startup code placed on actual chips.

As something that is implemented with hardware and also similar to software, code that is stored on the computer board itself (rather than a drive somewhere else in the computer or outside it) is known as "firmware." The operating system on your phone, for example, is stored as firmware.

The firmware for early computers (known as "BIOS" for "basic input output system") made it possible for the machine to locate and access drives and other devices, so it was possible for the computer to start itself.

Today, the BIOS is largely superseded first on Apple computers by the "EFI" ("Extended Firmware Interface") and then broadly by "UEFI", for "Universal Extensible Firmware Interface." It is common for people to still refer to these as the "BIOS."

The situation where the BIOS is most likely to be referred to is the "BIOS setup" mode, where it is possible to configure certain aspects of the machine even before any software has loaded.

When the machine is switched on, there is typically a screen that allows you to press a certain key (often one of the keys from F1-F12, or Ins or Del on the keyboard) that allows you to really mess with your computer setup.

Note that while learning more about the BIOS setup is useful, the BIOS setup is very powerful and if you change something without knowing what it is (or what it was set to, or how to change it back) you can do things like stop your computer from knowing it has a hard drive.

Since the operating system and your files are typically on the hard drive, you generally don't want to do this. Thus, exercise great care (and definitely take notes) when messing with the BIOS, if you choose to. There are situations where you want to change a setting-- pay attention and note your changes.

In most instances, if you accidentally change a setting while in the BIOS, you can back out of it and "Quit without saving." if you don't know what you're doing, this is advisable. Even if you do save arbitrary changes, you can often "Revert to defaults" on a PC. This is not the same as "factory defaults" on a tablet or phone-- it simply changes your BIOS settings to the ones that the machine came with originally.

In most instances, setting the BIOS back to defaults should be alright, and even fix the setting you made without noting what it was. In the rare instances where this does not work, an expert or professional can generally help you find and fix the setting. A cheaper and easier solution is just being careful and taking good notes.

Probably the most popular reasons to go to the BIOS settings are:

1. If you change the actual hardware setup inside the machine-- such as adding a RAM module. The computer may want to go to setup to save the change, but it will likely be automatic. You can also go to the BIOS setup to check if the change in ram has registered.

2. If you want to change the boot settings of the machine-- such as whether to boot from CD/DVD or USB first, or which hard drive (if you have more than one) to try to boot from first.

3. If your computer has a low BIOS battery, and loses settings.

The BIOS battery is a coin-like thing on the largest board in your computer. They are good for years, but an older computer could lose its BIOS settings when the battery fails.

Generally the computer knows this when you turn it on (the battery is not required to run the BIOS, just to keep the

settings stored on a chip and to run the hardware clock and calendar) and will tell you that the "BIOS settings are corrupt" or something like that.

Reverting to BIOS defaults is usually the fastest way to fix this (on each boot, particularly if the machine is unplugged) until you replace the battery.

Fiddling with the BIOS and batteries inside the computer may not be your cup of tea, even if they are common tasks for novices. These are essential skills in a computer shop, and many home users are familiar with them, though certainly not every user.

But your computer does have a clock that runs when the computer is off, it is powered by a battery, even when the machine is not plugged in, and you can look at the battery by opening the case and finding a coin-sized battery (it will be one of the shinier things inside the case) generally mounted flat on the largest board.

If you want to replace it, look up how online so you don't break any of the necessary clips, though it would probably be rare for a person to break it the first time.

After splitting computing into hardware and software and firmware, we can split it further into operating system and applications. We will not do this carefully, we will divide these with abandon and not worry about being too distinctive.

In practical terms, an operating system is a collection of software that runs your software. A fairly reliable, non-technical way to tell if a collection of software is an operating system, is that the author will tell you it is. This will rarely be untrue.

Once the system starts the boot (startup) process, an operating system generally picks up from there. The BIOS runs the bootloader program, which is usually a simple program that only loads the operating system, and then the OS does the rest.

It is possible to have more than one OS installed on your computer (known as "dual booting" when there are two, though I've never heard the term "triple booting" when there are three or more) and the bootloader will usually allow to select one before the operating system starts.

This is different from a virtual computer, where one OS is running inside another-- the bootloaders job is to boot just one OS per computer startup, even if it allows you to select it from a menu.

The "boot" or startup process for a computer is mostly automatic, and the purpose is generally to provide the

user with an interactive environment when the process is complete. The interactive environment can be one of the following:

1. A graphical, mouse or touchscreen-driven environment where the user types in, points and clicks, or touches elements of the system to use the computer.

2. A text-based environment-- normally what loads prior to the graphical one, either for maintenance or because the setup of the graphical system has failed in some way and needs to be reconfigured. In some instances the user may prefer a text-based environment and will only go to the graphical mode when (and if ever) necessary.

3. A machine that may have no monitor, mouse or keyboard, which is accessed by the network from another computer. A server is such a machine, and generally any modern desktop can be configured to run as a server this way, and your internet router is another such machine.

You can login to your router from your computer if they are on the same network, or in many instances if you are connected to it wirelessly. Once logged in, you can operate your router firmware from a text-based environment or most often from your web browser.

Note that any modern computer can generally run software that allows you to operate it from a text environment or web browser on another computer.

We have not truly delved into the numeric nature of all computing yet, but we have given an overview of the modern computer without much regard to brand, generation (these things are more or less true over the past 20-30 years) or configuration.

Numbers will provide our "unified field theory" of computing, but we have spent a fair amount of time looking at the surface and getting our toes wet before our deeper dive.

The majority of computer education is at the application level, which is possibly a great mistake-- applications have almost universal rules behind them, and on the surface they are very abstract.

Applications can be mastered through practice, but this can be done much more quickly and thoroughly by understanding computing itself. Application use will not always result in real computer literacy or greater self-reliance, but understanding computing most likely will.

We've talked about the most basic connections between the computer and software, and will talk more about both. If your

interest in the subject has increased, you are encouraged to find an inexpensive, 5-year-old computer for 100-200 dollars strictly to mess around with (including the BIOS!) Nothing beats hands-on learning, and you can use your regular computer to look things up and read this book. Also read:

https://freemedia.neocities.org/zero-dollar-laptop.html

Chapter 3: The holographic subject of software

One possible ideal for computing is to able to be able to run practically any software on practically any computer. This isn't very likely for tasks that require high-end processing power like video or 3D rendering, or even high-end video games. For tasks like writing code and web surfing, this shouldn't be so difficult.

Virtual computers are usually implemented in a language that runs quickly, though in some instances it is possible to implement a virtual computer in Javascript to run from a web browser, such as the DOS emulator that allows archive.org to offer old dos software running in a web browser.

In many instances, software can be translated, recompiled, or ported to other platforms. A more universal computing platform may or may not come about, but from the 1980s to the present there is an increase in both the number of platforms and the number of ways to make software cross compatible and thus run on a variety of those platforms.

Because it is available to everyone to use and share and modify, we will speak increasingly of GNU/Linux, though many of the things we will talk about are also possible from a

Windows or MacOS platform. A Macbook can (to some extent) run GNU/Linux directly, from DVD or USB, and Windows can run the Windows version of Qemu (which then allows you to try GNU/Linux from inside a window) depending on how many resources Windows leaves free.

If you are unfamiliar with GNU/Linux, one of the things we will be doing is taking it apart. But we will also be taking apart Windows and comparing it to MacOS, on the assumption that neither have changed enough over the past 10 years to make our comparisons invalid.

It is recommended that you go to Youtube and look at some versions of GNU/Linux, as well as MacOS, if you are a window user. If you are a Mac user, we recommend that you learn a few things about the command line in MacOS.

MacOS contains a powerful GNU shell, along with the Darwin (BSD-based) kernel, and (through its UNIX-based heritage) has more things in common with GNU/Linux than windows does.

As outlined in Chapter 2, the BIOS (or EFI or UEFI) hands the computer over to the bootloader, which hands the boot process over to the operating system.

The operating system runs an "init" and/or startup scripts to get the OS running and to present the user with an interactive environment (or, to run a server that can be accessed from another computer.)

It is at this point, between the operating system and the interactive environment, that we can begin to talk about user applications. And the purpose of doing so goes far beyond simply how to *use* the applications.

The interactive layer of the computer system is called the shell, and the shell runs applications.

For a command-line shell, the way to run an application is to type its name. For a graphical shell, you might open a window or menu that has a representation of the program-- an icon or a name for you to open by clicking on it.

A voice command processor such as those found in the speaker-bugs in your home also work like a command shell-- instead of typing the name, you say "Google, log that I'm now running my calendar program." Or you can say the same thing just by saying "Google-- what's on my calendar?"

Whether the program name is clicked on, typed in or spoken out loud, the shell recognises that the name is referring to something installed on the system, and it tells the computer to load that program.

Running a program and calling a function inside a program is a very similar thing-- a program may contain several functions, such as the **to_f** function we showed the code for earlier, and the function will "run" when the program refers to it by name.

Putting a file that has runnable code on the system will often let you run it by referring to its name. So using the command line is a lot like (or is) an act of coding.

When is a command less like coding, or a function call? When its a query. If you tell the computer the name of your browser, whether you type the name or click on the name or speak the name out loud, this is a lot like a function call. It runs the code referred to by that name on the system.

When you type a search into google (or into a filter-bubble-free search engine,) you are also calling a function. The function you're calling in this instance is a function on the Google servers to do a search of its data.

The data you type into the google search bar is called a "query," which is a bit less like coding, because you aren't referring to the function you want to run-- you are simply entering data which Google hands to the function or functions for you.

But ultimately, every process a computer (or internet website server) can perform can be thought of (or involves) a function call.

What is a function call? It is simply running a specific section of code that has a unique name.

The way the computer handles code is incredibly simple-- it has very few features that are fundamental to the task of computing. On top of these basics are sophisticated features relevant to performance and handling and addressing large amounts of information, but fundamentally the computer deals with numeric functions and conditional jumps.

These are the core features all modern CPU chips generally have in common. But lets make our way back to the graphical shell.

The graphical shell, like the computer itself, tends to deal with rectangles-- numerically, these rectangles may actually be straight lines:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

And the computer might not care if each discrete point in the line is arranged in a row, like this:

00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19

Or if the points are arranged in a rectangle, like this:

00 01 02 03 04
05 06 07 08 09
10 11 12 13 14
15 16 17 18 19

We think: 5 points across, 4 rows down-- the computer may think "20 discrete points." There is pretty easy math to convert back and forth either way. The way these points are numbered and referenced is called "addressing," like it is with post boxes.

But whether the addressing is in a line or rectangular, it is basically a fact that the graphical shell deals with rectangles. Your screen has a discrete number of points across, and a discrete number of rows down, and each point is called a "pixel," for "picture element."

When you click on certain areas of these pixels, the computer is told to call certain functions. This is the basic concept of how a graphical shell works.

The graphical shell is also called the GUI, or "graphical user interface."

GUIs have different design elements, such as arrows or a bar you can click on to scroll, menus you can click on that draw a box and put different items in a list for you to select, buttons you can click with the mouse to perform a certain task, or text boxes you can click on and then type text into.

GUIs can also give you pictures, drawing areas, checkboxes and other graphical controls.

In a web browser, the browser contents can be defined with document "markup language" such as HTML. HTML can be styled with CSS, and various aspects of the page can be defined, altered or controlled using Javascript. Using these languages, you can define the elements of a GUI within a webpage.

Sometimes, the code used to design a standalone application does not differ so much from the HTML equivalent. For example, a standalone GUI application may have the interface designed in XML, which is similar enough to HTML that some people less familiar with both may have trouble telling them apart.

With both graphical and text-based applications, it is generally possible to put data or interface elements on the screen using numeric values to represent the row and column positions for each element. If you are familiar with a number line, it is a simple form of graph:

```
< --|--|--|--|--|--|--|--|--|--|--|- 0 -|--|--|--|--|--|--|--|--|--|--|--|-- >
```

The arrows on either side represent that the numbers below 0 and greater than 0 continue beyond the range of the graph as plotted. Each vertical bar represents a value a fixed distance from the others, for example:

```
0 -|--|--|--|--|--|--|--|--|--|--|-- >
   1  2  3  4  5  6  7  8  9  10 11 12
```

While a Cartesian graph consists of two such number lines, including the other half for negative numbers and a vertical number line (also centred at 0) creating four quadrants of the graph, in computers the points on the screen are most often represented only with positive integers or whole numbers. (and most often including 0.)

```
0 --|----|----|----|----|----|----|----|----|----|----|---- >
 |                          (x values)
--
 |
-- (y values)
 |
--
 |
--        A typical computer graphics scheme
 |
V
```

Instead of using number lines, we can also use coordinates in a rectangle to represent the points:

```
(0, 0)  (0, 1)  (0, 2)  (0, 3)  (0, 4)
(1, 0)  (1, 1)  (1, 2)  (1, 3)  (1, 4)
(2, 0)  (2, 1)  (2, 2)  (2, 3)  (2, 4)
(3, 0)  (3, 1)  (3, 2)  (3, 3)  (3, 4)
(4, 0)  (4, 1)  (4, 2)  (4, 3)  (4, 4)
```

In this 5x5 "screen" we have 0, 0 at the top, and the rest of the points or "pixels" have coordinates at (x, y) where x is the horizontal distance in pixels from 0, 0 and y is the vertical distance.

In a Cartesian graph, (named for Rene Descartes) we would say that the computer screen occupies "the second quadrant" of the graph. The first quadrant is the upper right quadrant, the second quadrant is the one immediately below the first, and the third and fourth are to the left and then back to the top, in a clockwise order.

A difference between graphics and text is that in terms of addressing, it is common for the top-left corner to be addressed as (1, 1) rather than (0,0):

```
(1, 1)  (1, 2)  (1, 3)  (1, 4)  (1, 5)
(2, 1)  (2, 2)  (2, 3)  (2, 4)  (2, 5)
(3, 1)  (3, 2)  (3, 3)  (3, 4)  (3, 5)
(4, 1)  (4, 2)  (4, 3)  (4, 4)  (4, 5)
(5, 1)  (5, 2)  (5, 3)  (5, 4)  (5, 5)
```

So, if we are covering the top line of our 5x5 screen graphically, it will be with a line of pixels ranging from (0, 0) to (0, 4):

```
(0, 0)  (0, 1)  (0, 2)  (0, 3)  (0, 4)
(1, 0)  (1, 1)  (1, 2)  (1, 3)  (1, 4)
(2, 0)  (2, 1)  (2, 2)  (2, 3)  (2, 4)
(3, 0)  (3, 1)  (3, 2)  (3, 3)  (3, 4)
(4, 0)  (4, 1)  (4, 2)  (4, 3)  (4, 4)
```

But if we put the word "hello" at the top, it will probably be from (1, 1) to (1, 5):

```
h       e       l       l       o
(2, 1)  (2, 2)  (2, 3)  (2, 4)  (2, 5)
(3, 1)  (3, 2)  (3, 3)  (3, 4)  (3, 5)
(4, 1)  (4, 2)  (4, 3)  (4, 4)  (4, 5)
(5, 1)  (5, 2)  (5, 3)  (5, 4)  (5, 5)
```

Extra points if this stops you because you noticed that in the top graphical example, we have created a large block for each "dot" and in the second, text-based example, we have used characters such as "h" that have a shape which clearly requires more than a single pixel.

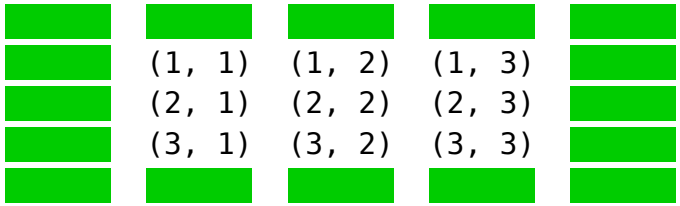
We can certainly do a graphical or "pseudo-graphical" display of graphical data using large character-sized blocks of the screen, but "graphics modes" on a computer generally have finer resolution (for example, HDTV does up to 1080 rows from top to bottom, and a larger number of units across) and text requires more than one row and column of pixels.

So lets say that each text character (along with the space between characters, and the space between rows, including room for letters like lowercase g, j, y, p, q...) requires an 8 x 12 rectangle of pixels. On a 1920 x 1080 resolution display, you'll get a maximum of 240 of those characters across, and 90 rows of text.

That display will let you show up to 21,600 characters of that size at once-- although on a 1920 x 1080 screen you may not want to use a font as small as 8 x 12 pixels, so we are probably going to display fewer than 21,600 characters.

At the moment this line is being typed, the screen is at a resolution close to 1920 x 1080, and the characters (if we take a screencap, zoom in, and count the pixel dimensions of this character:) are actually closer to 16x24. We should be able to get about 5,400 of these characters (1920 / 16 * 1080 / 24) on the screen at once, if our font is fixed-width.

Getting back to graphics, our example screen can hold up to 5 x 5 or a total of 25 dots, and each one can be any of up to 16 million colours. For this 5 x 5 box, we will just use green:



Ultimately, the means of drawing a box like this is to address each dot and change the colour information for each of these dots. Just like the location of each dot itself, the colour is represented and stored numerically.

Without getting into the way colours are converted into numbers and displayed as colours, we can say that everything that happens visually in a graphical interface consists of changing the values of the numbers that store the colour of each pixel.

Before, we said that everything that happens is a function call, and this is also true-- every function call references or writes the values of some of the numbers stored in the computer. To get information from a file to the screen, one function may read numbers from a file and another may write numbers to the circuits that change the display.

This is a slight over simplification, because of some of the other layers involved-- but fundamentally, this is what is happening when there are graphics (of any sort, including video) being displayed on the computer.

So the next thing to do is have routines that take information like dimensions and colour, and style and draw things like boxes and buttons and icons. These are also functions, and they are (fundamentally) similar from one computer to the next.

A lot of coding in everyday practice then has less to do with numbers then it has to do with data such as character strings, colours, and names of functions to call. But it is still numbers underneath all of this-- all numbers to the computer.

Chapter 4: Wasn't this about education?

So far, we have largely talked about computers, under a premise of talking about many things. We have explored computers from far above many of the details, with the hope that you might look for answers outside the order in which things are presented. Perhaps that is unfair.

An instructor can teach you to swim or drive, but they cannot swim for you. In most instances, they will not drive for you. But, an instructor can stand by and jump into the pool when there is trouble (taking the role of lifeguard) or grab the wheel in a car designed for student drivers, or a teaching instructor can help you like a tow service, to get your learning out of a ditch if you find your have headed into one.

Computers are a very broad subject, with many details to obscure the fundamental similarities. But computers are also an interesting starting point, because with a modern computing device you can move from "computing" to music, from computing to photography, from computing to art or writing a book, or remixing or creating video or text or anything else.

Computers can also be used for research, but having a search engine is frequently only useful if you know what to

search for. Except-- recursion is a computer concept, and so meta-application of concepts (and questions) is also the realm of computing.

If you don't know what to search for, you can search for "how to search" and learn more about search engines. Or you can pick up terms from a book, and search for those.

It is very useful, if you find a term that interests you, to look up that term. One of the reasons that most people are not great at doing online searches is that they have not deliberately practiced doing online searches.

If you set out to get better at searching for things online, you will get better at both creating search queries (I.e. what to type) and also following the results (what to click on.) These are modern information skills that often separate online "wizards" from online amateurs. Practice searching online.

But the role of guide (even tour guide) is useful, and can save you hours of searching. And as for this book, there are ways it can be improved-- here are some additional ideas on how to get more from it:

1. As mentioned, **do searches** on terms you find interesting. Practice searching. Learn about the "Filter bubble" and find a search engine that gets you out of that. Why don't we mention one? Because so many people know one already-- ask around, its free.

2. **Skim Wikipedia**. Few places online are more fun to explore without first knowing what you want to know. You could spend hours reading Wikipedia, or hours clicking through from one article to the next. If you don't like reading long articles, don't read long articles. Skim them for details and follow the trail of what you find most interesting.

3. **Just use Youtube**. For its mountains of problems, Youtube is the largest, most diverse collection of user-made and (sometimes) professionally-produced videos in all of history. You can find information on practically every subject.

4. **Remix**-- remix anything and everything. Remix just to practice remixing, but if you want to share your creations legally, you'll need to look for works that allow this. Fortunately there are many, and the number is increasing.

5. **Build a library online**. Build a library of your own works, or other works, or both. If the works are remixable commercially, add a CC0 license to your library and notify the free media alliance-- **https://freemedia.neocities.org/**

Our purpose is to build and foster the building of the largest, most user-friendly remixable library on earth (hopefully) and to strengthen the public domain. And to change education. Hey, it keeps us busy.

You really cant imagine the things you'll learn while putting a small library together. Librarians are some of the smartest, most formidable people in the world. The free media alliance is a partly-centralised, partly decentralised library. It consists of works such as this one, which you can remix and share.

6. L**earn how to code**-- whether Javascript, Python or some other language, learn how code works and how to do something practical with it. Please feel free to practice on fun, trivial, even decorative tasks (computer art is fun) until you find a practical idea that truly interests you.

Learning how to code for fun is basically the same process as learning how to code something practical. Computing all comes down to function calls anyway, as we will explain later when we talk about coding.

Coding will help you automate and partially automate many tasks related to research, learning, collecting, indexing, running a library or website, or who knows what else. We are not talking about "software engineering" or "developing professional applications." we are talking about simple coding., which is also the shortest route (faster than this book) to computer literacy.

7. **Teach**. Whether you are a professional teacher, a paid tutor, or simply helping out your friends, find an outlet for the knowledge you are most passionate about. Whether its a blog or website or book, write something. Create a video (this is one of the more popular ways to share knowledge today.) You can even create a class online, or teach one at your local library.

Also feel free to take this book and edit it to your hearts content-- find pictures from Wikipedia or Wikipedia Commons and create "The Illustrated Holographic Learning Guide" for this subject or others.

Release your new book (if you would,) under a Creative Commons license such as CC BY or BY-SA (please don't bother with -NC or -ND type licenses, they're relatively useless particularly for educational use.)

Of course we could delve into copyright issues in this book as well. Instead, we will say that this book you are reading is part of a global initiative to help move education and computing farther from monopolies, and closer to the domain of the user and learner.

We want your computing to be free, your software to be remixable, and your computer education to be complete and practical-- beyond superficial application training. We also want to build a large library of remixable works, including ones under the CC0 copyright waiver.

Why do this? Because then everyone can participate in computing and education as equals. Perhaps they will not be equals as an outcome-- there will always be experts-- but they will be equals from the beginning, with the freedom to excel as far as their natural abilities and interest takes them-- with no artificial restraints or limitations.

It is also nice to know that when you get your "zero-dollar computer" (we recommended spending 100-200 dollars on it, but this is frequently optional) you can start loading it up with software and books and watching free videos on it.

Along with the free media alliance, this book is also part of another movement called "OER" (for Open Educational Resources) to provide free and remixable textbooks to all students. There are certainly more conventional OER computing textbooks; this one is just intended to explore a different sort of learning approach along with computing itself.

Steve Jobs said programming "teaches you how to think." We wont go as far as that, but learning how to code and learning more about computers will teach you much more about how everyone thinks, how people learn, and how we create. So it is along these lines that we continue to talk about computing.

One of the cornerstones of holographic learning is the flipped nature of learning by doing what you don't necessarily know how to do. You learn to teach by teaching, you learn by teaching yourself. This is an innate human ability, and the goal of holographic learning is to assist and help sustain that ability, guiding it through some of the rough patches.

Chapter 5: The holographic subject of data

There are many different ways to describe data on the computer you're working with-- you could take a screencap the colour data for each pixel on the screen, or in a certain rectangular area of the screen. You could make a quick series of these screencaps and produce a video capture of you using the computer.

You could copy all the file data off the computer, creating a "snapshot" of the running system (at least some operating systems let you do this) and use it to produce a copy on DVD, to run as-is on another computer.

If there is a lot of text involved, you could isolate the text data and save it as a text file. Very many text files can be stored in a relatively conservative amount of space on a drive or folder.

But whether the application you're using is graphical or text-based, the data it works with is largely the same.

Graphical applications have certain advantages; they are generally considered better for drawing. They lend

themselves to video. In general, multimedia is better with graphics. When dealing with websites, it is often preferred to use a graphical system. Really great text browsers are rare.

Not every computer task however, requires a graphical system. This book is being written in OpenOffice, a graphical office suite available for several platforms. But most of this book could have just as easily been written in a simple text editor.

Graphical applications have advantages for someone who is unfamiliar or not entirely comfortable with a computing environment. They tend to be clearer on what tasks can be performed, with dynamic menus and visual controls. Many of these features can be implemented or simulated in a text-based environment as well.

For someone who has gained proficiency with the computing environment, graphical applications have a few downsides. One of the largest downsides in practice is the matter of screen real estate.

The more controls you put on a graphical application, the more space is taken up by the application. Several tricks are used to get around this-- from tabs for switching through "pages" of a dialog window, to panning or swiping from one page of a menu to the next-- though each of these loses the "everything in one place where you can select from it" advantage that GUIs have in the first place.

Coding graphical applications is often more work as well. With some notable exceptions, you simply need more code and more considerations to get a usable graphical application started and ready to use. Tools that help with these tasks include rapid application development suites, where instead of coding every aspect of the GUI you simply drop in and size controls, GUI displays, and other elements.

Then you write the code that makes them work together, using the same sort of code you might use to make any other application interactive.

Text-based applications can be fancy, but do not always need to be. Instead of implementing a control to enter information, you can simply add a prompt for each one:

What is your first name?

_

The line under the prompt is a cursor to let you know that you can type there, or that text is expected. In some text-based applications, you will have a search form similar to one from a GUI application, where you can view results while

you type in part of the name, and optionally select the rest of the name from the menu.

Most coders are still introduced to coding from a text interface, although it is increasingly popular to introduce coding concepts through a "drag-and-drop" graphical interface.

A possible disadvantage of this interface is not so much that it is easy to use (easy to use programming languages date back about as far as higher-level programming language abstractions do) but that the learner might not realise they are "really coding." some learners would not be bothered by this feeling, others could be discouraged by it.

Perhaps the nicest thing about text-based coding is the fluid nature of code. Generally speaking, "anything can do anything." if you want to be able to code:

"**baseball**"

and have that open a file window, you can actually have that. Just define your baseball function, give it some reasonable defaults, and call your function by name and it will do what you want.

Semantically, "baseball" is an absolutely terrible name for a file-opening function, that has no obvious logic to it. But it is a fine example of just how free you are to create a convention that works for you.

Similarly, you can create a picture of a frog and use it as the icon that starts your web browser. This too, has no obvious logic to it. But it is a sort of visual analog to creating a function with code. Perhaps a better function would be:

openfile filename

Where **openfile** is the function that opens your file-- and filename is the name of the file to open.

This can be a command (a function) you create with code, but by giving it a name you can "call" (use, run) the function at any place in the program.

define functioname parameters
[standard codes for opening a file goes here]
complete definition

In this pseudocode (pseudocode is made up code written to demonstrate coding without using real code) we have the top of the function definition: "**define functionname** parameters"

and we have a line that says "that was the last line of the definition" that just says "**complete definition**"

In some languages, that code may use punctuation and look more like this:

define functionname (parameters) {
[standard codes for opening a file goes here]
}

The braces **{ }** mean to "**start the definition**" and "**complete definition**". Other languages have commands for these purposes, such as "**complete definition**" or "**end def**".

The code to open a file depends on the language, and what to do with the file when its open depends on you. But by putting the code inside a function that you name, you are essentially creating your own "custom command" for opening a file-- and you get to decide what parameters (extra details) are important and vital, such as:

openfile filename

So code can very often be just as easy as that. Thus, sometimes when I am teaching people how to write code, the first thing I do is create a very simple language for them to learn coding with. It will have many of the simple features of coding included-- but not too many features, just enough to really get the idea (and then demonstrate it in practical contexts.)

specieslist = arropen "frogspecies.txt"

This is real code, in a language designed for teaching called "fig." An array is a group of places in the computer where it can "remember" data (at least while the program is running) and a file is (in this instance) a character string where each line is separated by a special "newline" character (or characters.)

You might type the names of several species of frog into a file and save the file, so it can be used later (or by another program.) Assuming you have created that file called **frogspecies.txt** (the .txt is a common abbreviation for "text" because that's the sort of data the file holds) then the code:

specieslist = arropen "frogspecies.txt"

creates an array called "**specieslist**" and loads each line of the text file into the array. If you stop the program, the array is gone. The file is not. But the array can be changed, added to, and even written out into a new file (or possibly, back to the same file it was loaded from.)

and this was done with one name (for the array) and one command, and one parameter. And code can be that simple sometimes. (A fair amount of the time, actually.)

When we talk about the simplicity of this function call-- because fig is a language made mostly by coding commonly useful tasks in the Python language, and then putting them into functions, with names that are easy to use-- "**arropen**" because "it opens a file into an array" -- certainly there is the more complicated Python code underneath.

But what we have done by creating "fig" on top of Python is, we have "abstracted" Python with an arguably simpler layer on top of it. fig is an abstraction of Python, which fig is implemented in. Python itself is an abstraction of the language "C" which is more complicated to learn and use. And even C is an abstraction of the computers own CPU logic.

Can you use an existing language to create a new language that can itself be used to implement another language? Yes! And that is the nature of abstraction.

When a language for coding is used to create a new language that is more difficult to use, or more obscure or peculiar to use, the resulting language is called an "esolang" or "esoteric language." I have also coined the term "easylang" for a language that is fairly easy to implement and also easier to use than the language it was created with.

If you stick to a small number of features, creating an easylang is easier than you probably think! You can also remix languages. For example, you may not like the abbreviation "**arropen**" for fig. You might feel that "**fileopen**" or "**openfile**" is better for the fig language.

It wont change the official version, but if you arent ready to create your own language and just want the command to be "**openfile**" instead of "**arropen**", you can get a copy of fig (written in Python) and change every instance of "**arropen**" in that text file to "**openfile**" and voila! You don't have to call it "**arropen**" ever again.

Although now the "source code" for your programs will be incompatible with everyone else that uses fig. They would have to change fig itself, or change your programs to be compatible with fig.

But all of this is possible with text-based coding. Of course, a lot of coding for graphical applications involves text-based coding as well. The difference is, when your application is also text-based, it might involve a lot less work (and less code) to go from starting a task to having working code.

And most of the time, when you are first learning to code,

you will learn text-based coding so you can focus on the coding, and not worry about styles, colours, widget layout, and other details that are peripheral to how code works.

As already said, some applications just don't lend themselves to text-based design-- most drawing programs do really well in a graphical environment.

You can "draw" with text, but it is much more limited-- even in colour. So text-based applications aren't everything, but they are a great place to start.

Once you are familiar with coding or using a text-based application, you will be a bit closer to understanding applications in general. There is an underlying simplicity in all this abstraction and function calling, where all tasks really just involve moving pieces of data from one place to another.

When you start to get a feel for that, you have discovered the essence of computing-- and now you know that all computer applications are really just doing this one thing: moving a bunch of numbers around in a very fancy, very sophisticated automatic calculator.

Chapter 6: A backtrack history of computing

I've spent years reading about the history of computing from many different sources, and a better work than this would list several different sources in a bibliography. Since I am writing this generally off the top of my head-- I did go to the Wikipedia page for Codepage 437 to copy the █ character, though in the days I used DOS I knew it as chr$(219)-- a bibliography isn't as feasible.

I have spent decades coding and written several long-form tutorials, which generally were "researched" by typing things into the computer and documenting what happened as a result (or documenting what I knew would happen, and then verifying it by running the same code) and I recently watched some of Part 2 of the series that inspired this book and the subject of holographic learning:

https://archive.org/details/buckminsterfullereverythingiknow02

And these talks which went on for 40 hours, were given by Fuller without notes. That is the style in which this book is written, but if the OER community is interested or has use for it then perhaps it will be transformed in some way.

Instead of starting with the abacus, this history of computing will begin with the present state of computing and work backwards, starting with the present.

There will be some amount of political bias here, feel free to "NPOV" those, if that's your thing:

Androids have taken over the earth-- at least, if by "androids" you mean devices that are run and programmed by Google, a large tech company that drives around everyone's town taking images of the fronts of their houses and other buildings.

AI, while considered by some to be impossible, unlikely or hoax-like (not entirely unlike the common sentiments about powered flight in the scientific community, prior to the Wright brothers) is nearly a household word.

Its capabilities include biometric recognition of a person by the front or back of their head--unlocking your phone with recognition of your face-- and recognising and manipulating images with an uncanny, almost human-like ability.

We are connected together by a global network of computers, which allows us to publish books at nearly zero cost (minus the work to produce the text and curate or produce the images) and you can run a free operating system on a five-year-old computer, which you got refurbished for a price of somewhere between 100 to 300 dollars, depending on the specs.

The long-promised video phone is actually a piece of software on your phone that is also a computer (or vice versa) and while for practically everyone that has spent the past 50-70 years expecting a robotic butler-- we still don't have those, but stores do carry now a variety of robotic vacuum cleaners.

Along with satellites, armed robots fly overhead and take pictures of us and occasionally kill people.

Public phone booths barely exist at this point, and both our phones and our pocket cameras (which can take enough video to produce a feature-length film) can also stamp the GPS coordinates where the video or photos were taken. And you don't even have to turn the camera around to use it, because its easier to just stick one on both sides of the phone.

For the most part, we have very few flying cars (I have not personally ever known an individual who has one) and very few self-driving cars, but both of these are produced in relatively small numbers. Its pretty futuristic here.

15 years ago in 2003, Youtube did not exist. Nor did the iPhone. Aaron Swartz was still alive. No one owned a Roomba. Javascript was much too slow (relatively speaking) for the sorts of applications people imagine creating or do create with it now.

20 years ago in 1998, Windows XP did not exist yet. Most computer monitors were based on large tubes instead of flat screens, except on laptops. Napster did not yet exist. The PATRIOT act, written by not-yet-vice-president Joe Biden-- did exist, but would not become law for three more years. CPUs were still sold based on clock speed, not by the number of cores (most consumer CPUs were single-core at this point.)

25 years ago, in 1993, the internet existed but gopher was more popular than the Web. Practically no one was using the Web anyway, and dial-up BBS systems were more popular. Windows was still DOS with Windows 3.x, since Windows 95 obviously did not exist yet.

30 years ago, in 1988, most people who had computers (including schools) were booting them with floppy diskettes. The software program and the operating system would generally share the same diskette, though sometimes you would use one diskette to start the computer and a second one to load the program.

Many computers had the entire operating system on a chip (particularly if it was just a BASIC interpreter and a simple command prompt.) Some of the programs might be graphical, but the operating system interface was generally text-based.

40 years ago, in 1978, it was still common to use a computer by dialing it remotely from a teletype-- a non-computing combination typewriter / printer / modem that simply sent codes for text characters to the computer through the phone lines, and then received characters and printed them out. UNIX and the C programming language were invented in the 1970s.

50 years ago, in 1968, UNIX and C did not yet exist. Mankind had not set foot on the moon. Computing was done on discrete circuit boards, rather than integrated circuit chips-- you could find and count the transistors (if transistors were used.)

The lander that would find its way onto the lunar surface in a year, had a computer that was being constructed out of a program encoded on wires threaded through magnetic cores-- this was similar technology to the "core memory" grids that IBM used. Tables and tables with many women seated at each were used to thread these cores by hand.

In the period between 1948 and 1968, computers went from

having programs encoded in language native to the processing unit, to having programs created in the first compiled languages. Grace Hopper wanted businesses to be able to use computers, and she knew businesspeople didn't want want to program the computers using just numbers, or numbers and mathematical symbols.

Hopper devised compiled languages based on English words, which was met with great scepticism. Today, most high-level languages are based on some kind of human-readable keywords, although punctuation is also used as program syntax for things like marking the start and finish of a loop or function definition.

Prior to this period, the primary uses of computers were to perform calculations for space exploration, weapons and other science and engineering-related tasks, and the demand for computers was small. They were used for the United States Census from the late 1800s onward to process census data, and they were used more ominously by the Third Reich to process demographic information about German citizens. In those days, computers were mostly electromechanical machines for sorting punched cards.

Reviewing forward-wise again with the history of computers in education, Grace Hopper pushed for higher-level compiled languages that abstracted the logic of the computer itself, and allowed the user to write programs with words based on English. This was the inspiration for the business language "COBOL". In the early-to-mid 1960s, this gave way to the educational languages "BASIC" and "Logo."

Another fairly popular educational language-- Pascal-- named for Blaise Pascal, was created. Javascript and Python and HTML would not exist until decades later, although Python had a language that influenced it called ABC.

Whether an educational language or not, Smalltalk should probably be mentioned as a language that changed programming forever. It also predates Javascript and Python.

In the 1980s, you were most likely to see computers used in primary and secondary schools for educational games and tutorials-- they were also used sometimes for programming in BASIC and Logo. By the 1990s, the previous decade's grainy, blotchy colour displays were being gradually replaced by higher-resolution monitors, which frequently offered at least 256 colours.

The 1990s also gave way to "multimedia" tools that mixed clips of low-quality video and increasingly high-quality sound, along with software becoming popular on CD-ROM rather than floppy diskettes, and "shareware" being software that you didn't own but were allowed to try and then either pay for

after trying, or ordering and getting additional, related titles or features.

In the 1980s, a project to create a UNIX-like operating system that everyone was free to share and modify was started, called "GNU" for "GNU's Not UNIX." By the 1990s, this system was being combined with a "Linux operating system kernel" that provided some of the functionality that the GNU system was missing.

In the early 2000s, it was possible to go to a store and purchase Red Hat Linux on CD (an operating system that was based on GNU/Linux) and a few years later, Canonical was shipping free CD-ROMs with their own brand of GNU/Linux called "Ubuntu" all over the world. Ubuntu was based on Debian, a GNU/Linux system that dated back to the mid-1990s.

Before the Web, internet use was not mainstream and consisted of the gopher protocol (popular with librarians and some universities) as well as FTP, USENET and email. The web (and the internet along with it) started to become mainstream from the mid 1990s to 2000 or so.

Within half a decade from there, smartphones and mp3 players and camera phones moved from being novelty items to everyday household objects. Tablets and changes in website design (particularly social media websites like Myspace, Twitter and Facebook) brought more and more people from a non-networked life to an online existence.

Before the 1950s, people were glued to their radios-- through the 1960s it was black-and-white television and by the 1970s more and more television was in colour. From the 1940s, the public went from mostly radio to television to computers in just 40 years.

Within another 20 years (just 60 years from the 1940s) people went from mainstream radio listening to mainstream internet use.

That's an unbelievable amount of technological change in the span of one lifetime.

Chapter 7: AI -- A futurist's interpretation of the present

One way to think of AI is "A lot more computing-- both good and bad." For art? Great. For surveillance? Sometimes bad. Apply it to everything-- people will. And it will be a great multiplier of things; of all computing tasks, more or less.

Not all at once. And I'm not hyping it, I'm describing the effect it will have-- as a multiplier:

Another way to think of AI is "enhanced computing." Because in many ways, it is fundamentally "just computer processing." anything a computer does is "just computing." But with AI, this becomes something more incredible. the scope of what can be touched with computing becomes richer-- for good and for bad.

Computing is very flexible, by design. We can actually say something about AI by while being this vague-- it is essentially like computer processing, except that it can do a little more, it can do more with more modest requirements-- it may take a while-- but with home computing equipment you can suddenly do things that you would expect of companies like Pixar.

Certainly not the resolution for a (feature-length) film like Pixar makes. They will still use large computing farms to get the job done in a reasonable amount of time.

AI can possibly seem to violate Moore's law, but it wont violate the laws of physics. if we are doing 1/3 of what our CPUs can do, then AI will make it so we can do the other 2/3 as well. And we can be really amazed at the results.

But also with "enhanced computing", things that once seemed incredibly difficult to program are now at least possible. Not necessarily "easy," but what once would take a team of 25-50 people (at least) can now be done with a team of 3. That's not a general rule, just that some things that once took many people can now be done with few, and faster than when it took more people.

Wizard-like stuff that once took a team can now be done by individuals. So the term "enhanced computing" is both telling and probably accurate.

And if you want, you can say that what computers could do already 10, 20 years ago is almost like magic. We know better, but for me it still feels a little bit like magic.

If you think of Harry Potter-- and I'm a fan of those books and movies, Ollivander said of Harry's nemesis: "He too did great things. Terrible, yes-- but great." It wasn't a compliment, it was an accurate measure. Of course for a young boy who just learned he was a wizard, it's creepy enough.

AI will do great things. Some of them will be terrible-- but great. And hopefully more of them will be Harry-like than Voldemort-like. But really, it will be both. AI is already used to help kill people. And I don't know how much we are ready for it. We should be cautious, and know that the best rules we come up with (like no doing magic outside the school grounds) won't be followed all the time.

No "Ministry of Artificial Intelligence" is going to be free of corruption or poor decisions-- nor would it be enough to stop all bad things that are done with or without approval. Either way, AI is here.

Perhaps the biggest difference between AI and human thought is the superficiality and bias. Humans have that sometimes, in very stupid ways, but we are more flexible. AI can magnify our stupidity-- think of the old adage about "knowing just enough to be dangerous." That's AI.

I'm not so much talking about "What happens if we copy people into AI versions of those people?" I'm really talking about the potential to try to make AI do what we think we want-- and getting far worse versions on average.

Because that's going to be very common, even humans have done this now and again throughout history. AI will lead us to

a greater capacity for such mistakes. Just as AI can solve things that would take 100 people to solve, it can make mistakes that would take 1000 people to create.

At least with laws, there's a judge and jury as long as its not artificial. We are certainly building corporations that have more power than a judge and jury do. But AI could do that too. My feeling about AI politically is still that it lends itself to many things, but it lends itself best (or at least most easily) to fascism.

That could be post hoc-- it's corporations and governments that are the most interested in it, so I could be describing what it lends itself to most easily by extrapolating it from the product of governments and corporations working on it. Still-- what are are developing now is like that.

People are trying to think of whether AI will be more good or more bad, and I'm not arguing for a neutral stance on it. I would say that if you look at all that computers have done both for our lives, and also to our lives, computing that is suddenly enhanced in ways that at least seem to go beyond the reach of Moore's law is exciting, but also justifiably scary.

What AI does is pattern recognition, and it can also impose patterns. This is said broadly because that's the broadness of the application-- you can find patterns similar to the way a person would, you can impose patterns similar to the way an artist would. Computers can do that without AI, but not at the same level as a person.

Today, we are designing software that can do those things faster and more tirelessly than people-- with similar (or sometimes superior) skill. Manipulating video, audio, tactile environments-- targeting, surveillance-- these are being expanded and developed all the time, not just in the future.

Chapter 8: Learning to code

We can be somewhat arbitrary and divide computer programming into 4 practical generations: lower-level coding, higher-level coding, object coding, and post-object coding.

This is not necessarily the way that programming languages are always divided, and it is probably a little misleading to do so. It would be unfortunate if people thought that lower-level coding is something people used to do. Although to be fair, now most people don't.

Lower-level coding is coding "close to the metal," where it is largely based on what the machine itself does. Machine language consists entirely of numbers-- one step up from there is mnemonics-- which are basically English-like abbreviations for CPU tasks.

From there we have various forms of coding, up until we get to objects. And then, says the fan of objects, we finally reached enlightenment and all things are possible.

The problem is, objects are a horrible place to start learning (in most languages) and you can do most things without them. I have watched for so many years, new people coming into coding and having fun learning to code, only to be

chided by fans of objective coding. If that doesn't sound like fun, you can think of objects as "like functions, only more complicated."

To be fair, the reason objects have so many fans is that they are really flexible and powerful. I love Python, and it gets a lot of its flexibility from objects (in Python, everything is an object.) But what I like best about Python (after the syntax and giant library) is that it makes thinking about objects optional.

Which is great, because using objects really does complicate a lot of simple coding tasks.

Fundamentally, the CPU has no concept of objects. They are not a native feature of the computer, but a concept. Most concepts in computing are closer to the basics of what a computer can do:

1. Count
2. Move (well, copy) data
3. Compare data and optionally go somewhere else in the code

We do just fine all the way up to functions, when a function is just "one or more lines of code, with a name and rules about what goes in and out of that code"

Objects are not as easy as that. With objects, a function is called an object method-- an object can have several methods, there is generally (but not always) some sort of class, and figuring out how to structure your objects is something a lot of people get wrong.

How objects work is a little different from language to language, which is pretty bad when they are complicated beasts as it is. Along with methods, an object can have attributes-- data that gets carried along with each object.

Coding is only as complicated as that if you want it to be. Proponents of object coding insist it is necessary, but that is where we come to "post-object coding."

Coding with objects is not the final word in computer programming. Like all other paradigms, objects have drawbacks that some people want to avoid. Objects are not always necessary. And coding with objects is harder to learn.

And this book will not teach you how-- rather this book will talk about coding from a perspective with a longer tradition-- where all methods are "functions" and variables are variables, data is data, and there isn't so much to worry about.

No one is denying you the opportunity to learn about coding with objects, however. If you learn Python, or Javascript or

even fig, any of these languages can be used to code with objects. (fig because it can do inline Python.)

If you understand how to code, that's a great time to go check out one of the countless online tutorials that will teach you how to create object classes, instances, methods and attributes. You might love it! We won't assume so, we want to make this easier than all that and it can certainly be done. So to introduce coding we can cover:

1. **Variables**
2. **Input**
3. **Output**
4. **Basic math**
5. **Loops**
6. **Conditionals**
7. **Functions**

A variable is as simple as giving a name to a piece of data. What makes it "variable" is that the data can be altered. A variable with a value that cannot change is called a constant. Python does not have constants, it is basically impossible to create a variable in Python that cannot be altered.

Some languages used colon-equals to assign a variable name (called an "identifier") with a value:

a := 5

In Python and Javascript and fig, 5 can be assigned like this:

```
# python
a = 5 ;
# fig
a = 5 ;
// javascript
a = 5 ;
```

A few notes about this code: in Python the final semicolon is never needed except between statements (this example has one statement, so the semicolon is only there because then it is like the other lines.)

In fig, the semicolon is entirely superficial and only there as a visual aid. In Javascript, every statement (even one on its own line) must end with a semicolon. That's just how Javascript is designed. It causes lots of needless problems, but it is a popular rule for some reason (C does it too.)

In Python and Javascript, the **=** between **a** and **5** is required, as it is with most languages. To demonstrate a feature of fig that is unusual, we will show the same lines with the minimally required syntax:

python
a = 5
fig
a 5
// Javascript
a = 5 ;

fig has required syntax, but very little of it. If you want to create a comment (a line of code that is ignored from being part of the program, or a part of the code to the right of the code that isn't ignored) you can use a hash:

a = 5 # this sets a to 5, but only the code to the left of # runs

Indeed, Python lets you do this too. So does Javascript, but it uses // instead of #:

// Javascript comment
fig comment
Python comment

A typical variable holds a single piece of data, referred to by its name:

a = "this is a variable"
b = "yes"
height = 5

Almost no language lets you use more than one word for a variable name, but you can sometimes cheat using underscores:

this_name_is_valid = "555-8282"

An array is a variable (arguably) that holds one or more pieces of data:

frogs = ["dart", "tree", "dancing"] # three values, one variable

A typical variable is referenced by using just its name, while part of an array is addressed by the name of the array and the index:

frogs = ["dart", "tree", "dancing"]
one_frog = frogs[2] # the third item in the array

This code copies one of the items in the array to a second variable, named **one_frog**.

Programming languages are often quite flexible-- we can even use a variable for the index:

frogs = ["dart", "tree", "dancing"]
which_frog = 2
one_frog = frogs[which_frog]

This code also copies one of the items in the array to a variable, named **one_frog**.

So that's variables. Now that we have learned one concept in programming, let's skip ahead to something harder (don't worry! We wont tell anybody you don't know this part.)

change_what = "new bicycles"
change_to = "baby kittens"
phrase = lineinput # the input command
p = phrase | split p, change_what | join p, change_to
now = p | print # the output command

Run this program, and it will wait for you to type something and press enter.

Type something random, and press enter. It will repeat what you typed. Now run it again and type this exactly:

There are 5 new bicycles at the pet store.

It will change this, and say back to you:

There are 5 baby kittens at the pet store.

Fig was designed for brick-by-brick teaching in the hopes of making learning to code absolutely painless. Since this book is about holographic learning, we taught one brick and then built a house with several kinds of bricks. Now instead of slowly and tediously building a house, we will explain how it was built now that you've seen it standing.

First, lets talk about syntax. Of our little program here:

change_what = "new bicycles"
change_to = "baby kittens"
phrase = lineinput # the input command
p = phrase | split p, change_what | join p, change_to
now = p | print # the output command

the only parts you need to type to make it work are:

changewhat "new bicycles"
changeto "baby kittens"
phrase lineinput
p phrase **split p changewhat** join p changeto
now p print

The only syntax fig requires is # hashes-for-comments and **"**quotes for strings.**"** Bold text is meaningless, underscores are just part of our variable names. The rest is to group code visually. Let's remove comments and add line numbers:

```
1 change_what = "new bicycles"
2 change_to = "baby kittens"
3 phrase = lineinput
4 p = phrase | split p, change_what | join p, change_to
5 now = p | print
```

Lines 1 and 2 just set variables **change_what** and **change_to** to the string to look for, and the string to change it to. The names are not special-- we could have called these **f** and **z**.

```
3 phrase = lineinput
4 p = phrase | split p, change_what | join p, change_to
5 now = p | print
```

Line 3 is the first one that contains a fig command. **lineinput** is one of fig's 90-something commands with a specific name, and what it does is get a single line of keyboard input. And it sets the variable to that input. So when we say:

phrase = lineinput

What we mean is:

"Create a variable named '**phrase**', but instead of setting it to a number or a character string, set it to whatever characters are typed in at the keyboard." right? That's easy.

```
asking = "what is your name?" ; print
yourname = lineinput
now = "hi, " plus yourname ; print
```

Sorry, second example there. Back to the first one:

```
3 phrase = lineinput
4 p = phrase | split p, change_what | join p, change_to
5 now = p | print
```

Line 4 does a lot... it copies **phrase** (what we typed) to variable **p**, it splits **p** into parts wherever **change_what** is found, and it joins those parts using **change_to**:

change_what is "new bicycles"
change_to is "baby kittens"

I type: "Play another record" it splits that by "new bicycles", you get:

["Play another record"] and you join by "baby kittens" you get:

"Play another record" (nothing changed.)

But if you type: "The store has 5 new bicycles."

and it splits that by "new bicycles", you get:

["The store has 5 ", "."]

And if it joins that by "baby kittens" you get:

"The store has 5 baby kittens."

So if you typed "The store has new bicycles, new bicycles, new bicycles!" it would change to:

"The store has baby kittens, baby kittens, baby kittens!"

Remember this:
p = phrase | split p, change_what | join p, change_to

is the same as:
p = phrase split p change_what join p change_to

Finally, line 5 lets us put the new phrase back on the screen:

5 **now = p | print**

Why is it called "print" anyway? You can also call it **sys.stdout.write()** if you don't like **print**, print is a very traditional command name that goes back to chapter 6, which says:

"40 years ago, in 1978, it was still common to use a computer by dialing it remotely from **a teletype**-- a non-computing combination typewriter / **printer** / modem that simply sent codes for text characters to the computer through the phone lines, and then received characters **and printed them out**."

...So the screen used to be a roll (or accordion-folded box-full) of paper. (Continuous-feed, accordion-folded paper was nice when it worked, but it was kind of a pain sometimes.)

Remember this:

now = p | print

is the same as this:

now p print

So we have a nice little program that does something fun and educational:

changewhat = "new bicycles"
changeto = "baby kittens"
phrase = lineinput

p = phrase ; split p, changewhat ; join p, changeto

now = p ; print

We promised variables, input, output, loops, conditionals, and functions.

Making a function is like making your own program command, so lets make a program command called changeit. We create our function with the function command:

function changeit
p = phrase ; split p, changewhat ; join p, changeto
fig

Here we say "create a function called changeit," we add a line of code from our previous program, and we add the "fig" command to say "get back to the rest of the fig program, stop messing around with this function definition." In Python we would just stop indenting, but some people hate that.

function changeit
p = phrase ; split p, changewhat ; join p, changeto
fig

There is a slight problem with our function: one of the nice things about functions is they have "scope". **Scope is sort of like Vegas:** "What happens in the function, stays in the function." Actually, it doesn't even stay-- it just doesn't go anywhere else.

The function is isolated like a miniature program, and this has one really major advantage-- it means that if we use a stupid variable name that we are using 5 pages up or down from here, it doesn't matter-- variables inside the function are totally separate. So if we have a variable called "hairdo" somewhere else, the variable "hairdo" we use inside our function wont "mess" with it.

We just have to get the program to talk to the function by adding parameters:

function changeit (phrase, changewhat, changeto)
p = phrase ; split p, changewhat ; join p, changeto
fig

Instead of **now = p ; print** we will help **p** back out of the function by "returning" that one value using **return**:

function changeit (phrase, changewhat, changeto)
p = phrase ; split p, changewhat ; join p, changeto
now = return p
fig

So what we've done here is incredibly powerful-- we have created a programming language command!

Our command is called "changeit" and it takes three parameters: the phrase to change, the part to look for, and the part to change it to. We can "call" the function like this:

now = changeit "haha" "ha" "he" ; print

Instead of saying "haha" it will say "hehe". We can recreate our first program like this:

phrase = lineinput
now = changeit phrase "new bicycles" "baby kittens" ; print

So we have introduced functions. We can also put this code in a loop:

while
phrase = lineinput
now = changeit phrase "new bicycles" "baby kittens" ; print
wend

Now it will wait for you to type something and change each line you type, repeatedly. We can make it do that just 5 times:

for p = 1, 5, 1
phrase = lineinput
now = changeit phrase "new bicycles" "baby kittens" ; print
next

The important thing is that when you write this program, you have to include your function definition:

function changeit (phrase, changewhat, changeto)
p = phrase ; split p, changewhat ; join p, changeto
now = return p
fig

in the code to be able to call your **changeit** function. But once it is defined, you can call it from all over the program like a regular command.

This function is in the public domain, so you can use it freely in any program you want. In Python, this is called the **replace** command.

We have introduced:

1. variables 2. input 3. output
5. loops
7. functions

We have not covered basic math or conditionals, though we used **plus** to add two strings.

Our loop worked like this:

1. **while** goes at the top.

2. **wend** goes at the bottom.

The stuff between those commands keeps running.

Conditionals work similarly:

1. **ifequal c d** goes at the top.

2. **fig** goes at the bottom.

The stuff between those commands runs only if **c** and **d** have equal value. Also they don't have to be named that:

phrase = lineinput
ifequal phrase "hello"
now = "hello yourself!" ; print
fig

Basic math...

now = 5 times 2 plus 11 divby 7 ; print

Variables, input, output, basic math, loops, conditionals-- and functions!

Those are the 7 programming concepts fig was created to teach.

Did we cover them too quickly? Should we give more examples? A book just about coding might help, though we are providing this here for other comparisons to computing.

Our example takes input and creates output, it works with simple variables and uses them to create arrays-- then it joins arrays with another variable and creates simple variables again.

Arrays are indexed sets of data. The computer also has an index of things it can address-- from the dots on the screen to the numbers stored on the hard drive. For example, in ASCII coding, the letters that spell "hello" are numbers:

ASCII 104 = "h" (...binary 01101000)

ASCII 101 = "e" (...binary 01100101)

ASCII 108 = "l" (...binary 01101100)

ASCII 111 = "o" (...binary 01101111)

So in binary, "hello" can be represented as **01101000** 01100101 **01101100** 01101100 **01101111** or in decimal: **104** 101 **108** 108 **111**.

And the computer can put those values 01101000 01100101 01101100 01101100 01101111 somewhere in RAM, and when it reads them and sends them to the display, you will get "hello."

As we have already demonstrated, you don't have to encode the binary yourself. Simply give the computer "hello" and fig will translate that to Python code, Python will run that using C code (or other code) and the C code is compiled to numeric codes the computer understands directly.

Although it is an incredible feat, computers started out as coded in numbers the computer understands, and eventually compilers were written to translate less CPU-like instructions into CPU instructions. Compilers take input, just like our example did, and change the values in it until the instructions we give are instructions the CPU can understand.

For example, I did not hand-code the letters in "hello" into binary. I ran Python and typed:

>>> **for p in "hello": print ord(p),** # and it replied:

104 101 108 108 111

>>> **for p in "hello": print bin(ord(p)),**

0b1101000 0b1100101 0b1101100 0b1101100 0b1101111

Once the translator is written, the computer can automatically translate very specifically-designed codes we can understand into codes the computer can understand.

Or another way of putting this is-- compilers (and interpreters) are programs that teach the computer how to understand new examples of code. Each abstraction above the numeric CPU chip instructions either has to translate directly to those instructions-- or to an intermediate layer (another program) that can translate to those instructions.

CPU <- C compiler <- Python interpreter <- fig translator

The C compiler can translate itself (this is called "self-hosting") into numeric CPU instructions.

The Python interpreter is implemented in C (Python is actually implemented in several languages, there are several versions of the Python interpreter. One version is implemented in Javascript so it can run in a browser.) And fig is implemented in Python. fig works by taking codes written in its own language set, and translating that to Python code, which you can then run learning Python.

But because of these handy abstractions, you don't need to know Python to code in fig. You don't need to know C to code in Python. You don't need to know CPU instructions to code in C, although one of the key lessons in this book is-- learning a bit about the layer beneath the abstraction you use greatly helps you to understand the layer you work with.

So if you learn a little about the CPU, you'll be a far better C coder. If you learn a little about C, it might make you a better Python coder. (Python is one heck of an abstraction-- but not always. Some of the syntax is clearly inspired by C.) And fig allows inline Python code:

```
python
print "look, this is Python code! " * 5
fig
```

And learning Python would let you create or modify your own programming language, based on or inspired by fig. It will certainly make you a better fig coder, and fig is designed to help you transition to Python (if you want to.)

That example can also be done purely in fig like this:

```
now "look, this is fig code!" times 5 ; print
```

And learning coding will generally make applications more obvious to understand-- so that is true computer literacy.

Everything in the computer is addressable through numeric values. Although some details (for security, for practicality) are a little more complex than "everything is a rectangle or list of numeric addresses," it's still useful to think of it that way. There are addresses for video output, for audio, for hard drives, USB and DVD, and how these are abstracted depends on the OS implementation. But fundamentally these are similar, sometimes identical-- from platform to platform.

Chapter 9: The holographic subject of hardware

The idea of this type of learning is to avoid the detail-heavy, very-specific-goal sort of introduction and give a broad foundation with lots of directions to take your exploration in. If you cannot afford a second computer to experiment with, you can learn to experiment with a virtual computer.

The trick is to have enough resources available (Windows for example, tends to run a lot of stuff in the background so that if you want to run a second operating system, you may need to learn how to make the resources it is using available to the second OS first.)

You can learn a lot about computers even from a virtual machine-- you can even learn to write your own operating system for one. But you may take interest in repairing or upgrading physical machines, and this chapter is a beginners guide to the concept of troubleshooting and repair.

Coding can be made a lot easier than repairing hardware. With hardware, details become unavoidable. But you can still start with generalities as your foundation, and get a better background than people who start with the usual mountain of details.

Debugging code is a little easier than diagnosing hardware

too, because half the time your code isn't working, it's something you did-- your mistakes will probably be common, whereas with a computer it could really be anything. Also with code, you can just make a copy and experiment and it won't hurt anything. When you do physical repairs, trial and error could actually break something in a way that you have to throw it away (or it becomes too costly to bother fixing.)

This chapter won't teach you how to repair computers, but it will introduce some concepts. Don't worry if there are words or ideas that are difficult to understand or keep track of-- just treat this chapter like a story, and follow the story as best you can. It's hard to guess if we can make this easy, or perhaps it will be very easy.

If this chapter fails in that regard, keep it easy for yourself by not taking this chapter too seriously. You can always pick up a study book for the CompTIA certification, which will introduce you to the mountain of details that the certification is based on.

Holographic learning is about starting with the whole of the subject and taking it apart, rather than building a tiny piece at a time. We have painstakingly introduced computers in the broadest terms, as machines that move numbers around-- to hardware devices, from hardware devices-- everything is encoded into numbers by digitisation, and then dealt with as numbers by the main CPU chip. Before integrated circuits, the CPU would be built with individual (discrete) transistors or other digital components.

Analog computers are also possible. Instead of dealing with on/off and 1s and 0s, some computers deal with variable voltage levels. Digital computing is probably easier to explain and more efficient and cheaper for the myraid uses we apply it to, but analog computers are not impossible to build.

Before the transistor was invented in the mid 1900s, computers were able to perform logical operations using vacuum tubes and relays. A vacuum tube works a lot like a transistor, in that you can control the flow of electricity using another flow of electricity.

Instead of a wire that lets electricity through at a more or less constant rate, a tube can let electricity through based on the rate at which the control circuit has electricity flowing through it. So it is a bit like an electrically-controlled switch. Transistors make this possible also, except they can be made a lot smaller, they run cooler with less electricity use, and are cheaper to mass produce than tubes.

A relay is also an electrically-controlled switch, except it is easier to explain. You can make your own relay fairly easily, to experiment with digital computing without computer chips. You can even do some basic math using 5 relays, 5 small lights, a power source, 5 push-button switches and some wire.

To make your own relay, the first thing you need is a spring-

loaded contact. You can have a lever that a spring keeps from touching the other contact, like this:

```
 ~((((((()~./  |------- contact 2
 contact 1
```

You can try using the spring as the contact, or you can solder a wire directly to the lever. The important thing is that the level cannot touch contact 2, because the spring holds it back. The conductors that go to the lever (contact 1) and contact 2 are the posts (or wires) for your switch.

Now coil a wire around contact 2, so you are building an electromagnet. This magnet will attract the lever, and when there is enough magnetism (when you power the electromagnet) it will close the switch. The two ends of the coiled wire are your posts (or wires) for the relay control.

So you have two wires that power the switch, and two wires that are connected when the switch is powered. You can do digital computing with just electromagnetic relays and push buttons and lights, if you are that interested in the history of digital computing and want to design or understand digital circuits. But relays, tubes and transistors are all switches that are controlled by electricity-- they are controlled by electricity, and they also control electricity, because they are switches.

And electronic digital computers have generally been made out of relays and tubes or transistors, which have this functionality in common. The earliest automatic computers were powered by rotation and used gears, including those old hand-cranked calculating machines.

But getting back to modern transistor-based computers, it is worth considering how similar each type of computer really is:

Desktop:

1. housing and power supply
2. internal storage (drives) and external storage (USB drives)
3. external keyboard and pointing devices and screen
4. CPU and RAM and firmware chips, network connection
5. peripheral (external) devices and internal add-on cards

Laptop:

1. housing, portable battery, external (wall) power supply
2. internal storage (drives) and external storage (USB)
3. integrated keyboard (optional external too) and touchpad or other integrated pointing device-- you can add external versions if you want to (or if the keyboard wears out or is damaged) though these options are less portable.
4. integrated screen. Like with other devices, if the screen breaks you can use an external at cost to portability.
5. CPU and RAM and firmware chips, network connection
6. peripheral (external) devices and internal add-on cards (though considerably smaller add-on cards)

Desktop: (server configuration)

1. housing and power supply
2. internal storage (drives) and external storage (USB drives)

3. external keyboard and pointing devices and screen, which are often not attached. Instead, the network connection makes it possible to use the keyboard and pointing device of another computer, to control the server through a networkable interface (either a text-based or browser-based environment that the server makes available over the network.)

4. CPU and RAM and firmware chips
5. peripheral (external) devices which are probably not used, and internal add-on cards

Laptop: (server configuration)

1. housing, portable battery, external (wall) power supply
2. internal storage (drives) and external storage (USB)
3. integrated keyboard (optional external too) and touchpad or other integrated pointing device-- generally not used except for setup; not that laptops make great servers.

4. integrated screen. CPU and RAM and firmware chips, network connection works like the one in the desktop server.

5. peripheral (external) devices and internal add-on cards (though considerably smaller add-on cards)

Router or modem:

1. housing, usually no battery, external (wall) power supply
2. often no internal or external storage, except for firmware, which can often be overwritten (very carefully.)
3. no keyboard, pointing device or screen-- usually has status lights, network connection or a few specific buttons provide the control interface (runs like a server-- text-based or browser-based controls.)
4. CPU and RAM, firmware was already mentioned.
5. generally no peripherals or add-on cards.

Single-board computer: (SBC)

1. often no housing (DIY or buy one separate) and no battery, external power supply
2. some have internal storage, OS is often written to an SD flash card (this is more convenient than firmware.) USB storage devices are often possible. Bootloader is sometimes/often stored on a firmware chip.
3. often includes USB or and/or wireless support for keyboards and pointing devices and other peripherals. Often includes HDMI port, sometimes includes composite (video) port, rarely includes other video ports like VGA or DVI. Sometimes includes integrated WiFi support, often includes support for network (Ethernet) cable. Some models lack video and can be controlled through network interface like a router-- most models can be setup/used like a server anyway. **4.** CPU and RAM. **5.** external USB devices, some models have (relatively limited) proprietary add-on cards

Netbooks:

Exactly the same as laptops, except smaller. Early netbooks were often price-subsidised by wireless (cellular) network contracts, the way smartphones are now.

Phones, tablets, phablets and mini-tablets, some extremely fancy MP3 players:

1. housing, portable battery, external (wall) power supply
2. sometimes internal storage (drives) especially solid-state
3. touchscreen provides functionality of pointer and screen, virtual keyboard or bluetooth for other inputs. Some support USB devices.

4. CPU and RAM and firmware chips, network connection is usually WiFi and often includes wireless (cellular) network support. It is possible to run a server from such a machine-- it is most common to share the wireless connection, but you can sometimes login to one via text or access files via web browser on a different machine on the same local network.

5. peripheral (external) devices (usually wireless)

Although many of the smaller devices use ARM processors and many of the larger devices use Intel/compatible processors, all of the above are examples of computers with more in common than different.

A tablet with support for USB or wireless keyboard and mouse and a connection to an external display could still be used as a desktop, even if it would make a really lousy desktop machine. An SBC can be used as a router or server, although it would be very low-power and probably relatively low-bandwidth.

A router could be used as an SBC, though getting firmware for it to do much that's very useful is a bit of a chore, and it still won't offer a display except through a second computer over the network.

A server can be used as a desktop but you may have to add a video card (they're often included) and an old desktop can be used a server, though commercial server usually have specs optimised for operating as a server.

By far, the easiest of these to take apart and modify is the desktop, so lets get an imaginary computer (or a zero-dollar one someone has laying around) and open it up.

The first thing you want to do is figure out how to open the case. Usually one panel comes off the side. If you remove an entire side panel and find a lot of shiny silver dots, that's the wrong side-- put the panel back on and remove the panel from the opposite side (many cases don't open on both sides.)

Some people are worried about injuring themselves if they open the computer-- you are far more likely to injure the computer (that's why we are using an imaginary or zero-dollar one.)

If you are not experienced with electronics, open the computer with an experienced person supervising or get a book that goes into all the proper safety procedures. Here are the main things to consider:

Assuming there isn't a dangerous short (anything in a house or building can have a dangerous short. Some idiots install wires so incorrectly that they run electricity to the bathtub or shower. This is extremely dangerous, and it goes without saying that bathtubs and showers should not have any electricity going to them at all) then most of the inside of the computer should be safe-- for you.

The parts that are dangerous are usually in a large metal box that have bundles of wires coming out of it. If you open the case and think: "Alright a there are a lot of metal boxes that have wires coming out of them" we are probably talking about the largest one, and the one with the most wires.

But another thing to be aware of is that most of the case is made of fairly thin metal. If you look on the outside of the case where you plug the power cord in, this gives you a really good idea of which box inside the computer is the power supply-- its the one directly on the other side of the same part of the case. This rule applies to anything else that has features on the inside and outside. Which card is the network card? It's the one that allows you to plug a network cable in on the other side. But that plug is often built into the main computer board.

Do you have two places to plug in video? If they are side by side, one is probably VGA (it looks like this: \`::::/ except instead of two rows of holes it has three) and the other is probably DVI.

DVI is larger than VGA and rectangular, and is worth looking up so you have a picture (you can do that with any of the typical connections too. Just type **en.wikipedia.org/wiki/** and then add DVI after it, to look up DVI. Or do a search, it just takes longer sometimes.)

You know what HDMI looks like, if you ever use the cable for one. If it fits an HDMI cable perfectly (never use force for cables, you'll break stuff-- be gentle) then it's an HDMI port.

But getting back to safety concerns-- ahem!

1. older computers contain lead solder. Do not touch the solder. After handling old computer parts, radio parts, electronics, even the wires of cheap headphones (yeah, it's awful) wash your hands with soap. You don't have to scrub so hard that you break the skin (rather better if you don't, actually) but do rub the soapy water against your hands to clean them. Cheap PVC insulation can contain some amount of lead too, hence "cheap headphone wires." Good wires

have a logo that says "ROHS" (which may amuse fans of *The Princess Bride,* even if it's "U" not "H") although some ROHS logos *could* be fake/counterfeit. ROHS means "reduction of hazardous substances," and often points to reasonable quality headphones, among other things.

(Though many ROHS headphones won't cost more than 20 dollars, and you're certainly not going to get top-of-the-line quality for that no matter what Bozo tells you.)

By "older computers" this generally means "before Pentium" so if you have a P4 or newer (in other words, if the computer was made after 2008) you should be fine! But if you don't know the vintage of each and every board, perhaps you can still wash with soap and water before having a sandwich?

The power supply (on the other side of where you plug the computer in) -- don't put anything inside it, whether it is conductive or non-conductive, do not disturb it. If there is something wrong with the power supply, most people do not fix them; they are better off replaced. You can remove the power supply, you can take if off and handle the outside, but do not open it or poke things into it. Just pretend it's a toaster.

Don't worry about bumping into the power supply though-- the outside of the power supply has a direct connection to the entire computer case for grounding-- if the outside of the power supply were going to hurt you, touching the outside of the computer case would probably hurt you too.

It goes without saying that you shouldn't smoke or have liquids (or most aerosol sprays) near the inside of the computer. If you have a beehive hairdo-- or long hair like I used to have, do not put it inside the computer (for the computer's sake as much as yours.)

The main concern for the computer is static electricity (and pulling things out or putting them in while the computer is turned on-- don't do that, you could damage practically any component of the computer that way.) The most common ways of reducing static electricity damage is:

1. don't work on carpet whenever it's avoidable (good idea, not enough by itself.)
2. wear anti-ESD shoes (these prevent electrostatic discharge which can damage the machine when it is open)
3. if you don't have anti-ESD shoes, at least touch the outside of the power supply (with your hand) as it is grounded. This will reduce the likelihood of ESD damage.

Also: never handle components by or touch the conductors on the edge connectors. Avoid excessive handling of components altogether. And consider getting the shoes, they're actually very comfortable.

Just to repeat: don't pull components out or install them while the computer is on. It will very likely crash the machine, and circuit damage is also likely.

With the exception of cable attachments, most of the things

inside a computer you'll need two hands to install or remove them. Some have latches that need to be unlatched first, and they are *usually* obvious (one one side of the device, or both) if you look closely. Sometimes they are hidden if another component is in the way.

Putting a component in halfway or crooked can damage the circuit, prevent operation or at least cause intermittent operation.

During a move or shipping, add-on cards can sometimes get loose, and need to be re-seated before they will work properly. Hitting the computer probably won't re-seat them, so avoid the temptation. That might work with oxidised or loose connections on old TV sets, but it's bad for the hard drive (and might unseat other boards in the process.)

Desktops tend have to well-engineered parts and good connectors, on average. (Sadly, this does not always include the power connector. Or occasionally, some USB connectors.) If you are gentle or look up the proper way to handle the connector, working on a desktop is relatively easy.

Laptops are another matter. The nice easy-to-remove ribbon cables on desktops are often replaced by flatter ribbon cables with connectors that are stupidly easy to break and the ribbon can even tear. This, for a ribbon that is probably specific to that model (most desktop ribbon cables are standard.)

Laptops and tablets and pretty much anything except desktops are full of plastic clips that are easy to break. When you break them, the things never fit together exactly the same again. They are generally usable, but it's never as nice as when the clips are all intact.

These things are also held together with screws, but the clips form a more complete fit. Sometimes glue is the answer. Generally speaking, glue is NOT the answer! If you are grading for glue (other than glue made expressly for the specific area where it's being used) then using glue ranges from a C- to a D... to an F if it damages anything.

(If you're a perfectionist and you broke a clip, don't feel too bad. A good professional generally avoids this, but everybody's done it once before. If you have a habit of breaking clips, you might want to learn how to open things properly before you go opening up a shop.)

We are having a lot less fun with this chapter than the others, because we are trying not to (literally) break stuff. And since we have spent most of the time talking about how to protect the computer from you, other than "don't poke a toaster, and wash your hands after handling lead" (and dry them well before you work on anything else, and put on a jacket if it's chilly outside...) you know there are obviously more ways for you to hurt the computer-- unless you drop it on your foot or have trouble carrying it.

But now that we've got all that out of the way, desktop

computers haven't changed all that dramatically in 30 years.

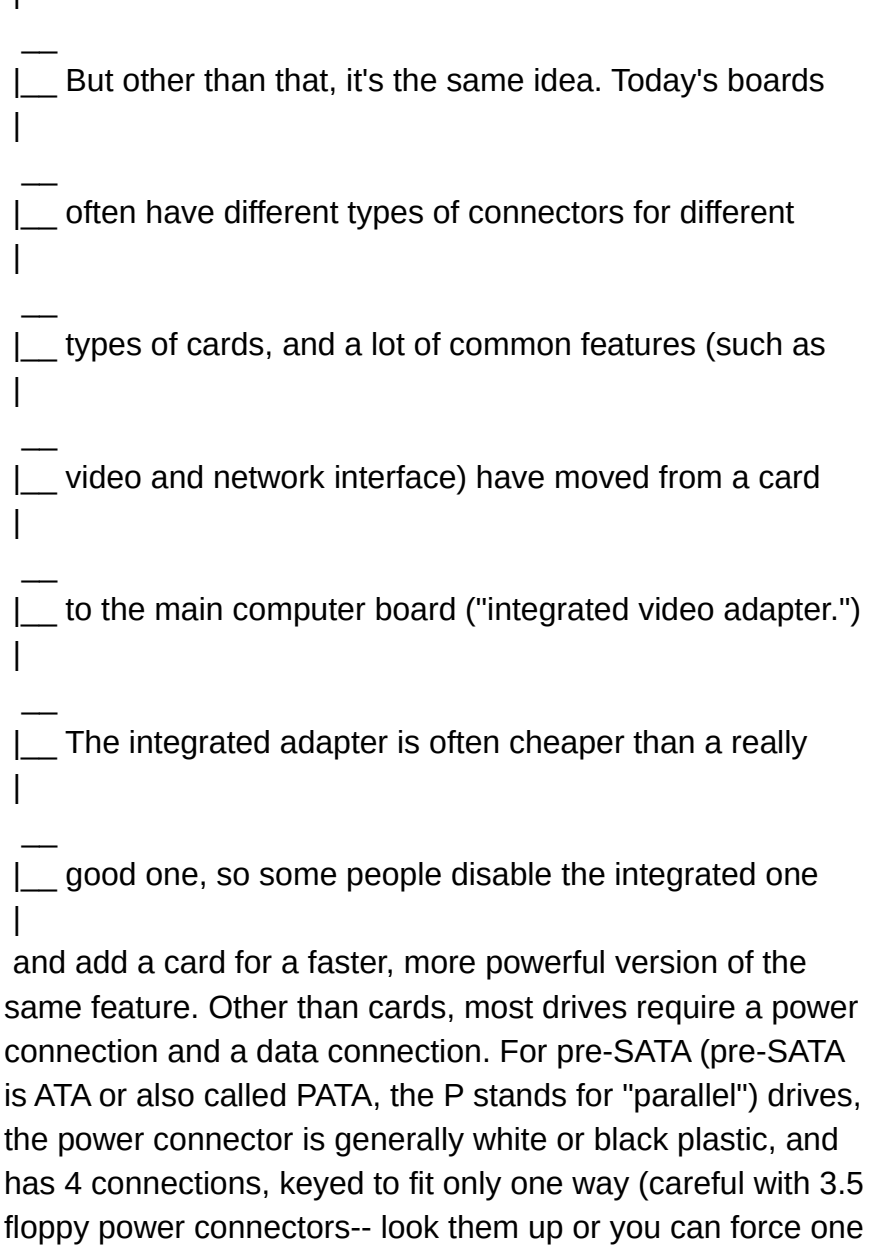
An old bus 100 main computer board had a bunch of connectors to plug cards into, like this:

Just like an old video game console, the cards plugged vertically into the board. What's a card? A card is a board that you plug into the larger board. Unless the card is really small, then sometimes it's a "module."

Those bus 100 boards are ancient-- you would probably never find one accidentally (I've never found one.) But they were very simple. You plugged cards into them, and then connected other things on the boards with wires.

My first PC was similar-- a board (horizontal) with cards (vertical) and some wires between them. The Apple IIe was like this (with an integrated keyboard.) And if you wanted to add a peripheral, you might open the computer up and plug a card into the board, rather than using a USB connector.

For a tower, the board is mounted vertically, and the cards plug in horizontally:



But other than that, it's the same idea. Today's boards often have different types of connectors for different types of cards, and a lot of common features (such as video and network interface) have moved from a card to the main computer board ("integrated video adapter.") The integrated adapter is often cheaper than a really good one, so some people disable the integrated one and add a card for a faster, more powerful version of the same feature. Other than cards, most drives require a power connection and a data connection. For pre-SATA (pre-SATA is ATA or also called PATA, the P stands for "parallel") drives, the power connector is generally white or black plastic, and has 4 connections, keyed to fit only one way (careful with 3.5 floppy power connectors-- look them up or you can force one

upside down and that's definitely bad... most connectors won't fit upside down and the floppy power connector is supposed to fit one way, but it's too easy to force upside down.) And with these older drives, the data connection is larger (more pins) and the power connection is smaller (4 large pins.)

With newer SATA (S for "serial") drives, the larger connector is actually for the power, not data. Don't worry about that though, the data connector never fits the power connector-- and vice versa. You won't mix them up except when you're trying to name them.

On older PATA drives, the desktop-sized drives and laptop-sized drives use different connectors-- they're not compatible without an adapter, so you can't just plug a laptop drive into a desktop. With newer SATA drives, the big desktop versions and the mini laptop-width drives use the same connection-- you can take the drive right out of your laptop and install it in a desktop.

(You'll have to be creative in the way you mount it, as most desktops do not include brackets for mounting a laptop drive. If you buy an SSD drive, which is SATA and normally laptop-sized, it will often come with brackets for mounting in a desktop.)

If you creatively mount things, it is good to not rely on tape (it can detach) or anything that make it so that metal components are free to swing to either side (and short out a card or something.) But you can be creative, as long as the component is secure. Proper grounding is always a plus!

It's also worth noting that many people will put computers on a board, while the board is a non-conductive surface. Wood is sometimes used, it is a slight fire hazard-- cardboard should be avoided, some companies make mats for working on a computer when the main computer board is not mounted to anything.

Note that when a computer board is properly installed, the entire case helps to ground it-- this is better, and cases reduce RF noise/interference as well. Routers and SBCs are usually mounted in plastic cases, (sometimes made of Lego) and the parts that create the most RF sometimes have metal shields on them.

Someone told me recently that their laptop is having problems with the battery. Unlike tablets and phones, laptops will generally work plugged in without the battery installed. If you try this with your phone or tablet, it will probably just shut off after finding out that it can't charge.

But a laptop will work plugged in without a battery-- it's like a small desktop at that point. If your power lines are not reliable, you can get a UPS to plug it into.

So to diagnose the battery problem on the laptop, simply power the machine off, remove the battery, plug the machine in, and use it without the battery.

If the battery really is the problem, this will solve the problem (recognising of course, that now it is not portable.)

However, most (perhaps not all) problems with the battery will stop the moment you plug the laptop in-- since I'm told the laptop powers off randomly even when it is plugged in, I do not think it is the battery. If it is, it can be solved the same way-- remove the battery and work with the power cord only.

But that would mean the battery is the cause of it shutting off even when plugged in... which makes me think it is shutting off for some other problem. (A short, or a broken connection, or a chip that is overheating/failing.)

It would be nice if this were just a battery problem. Unfortunately it sounds like the laptop is dying in a way that is too expensive to be worth fixing. A desktop would be easier to salvage. The drive almost definitely can be-- it could be added to another laptop, or to a desktop.

The drive does contain Windows, and Windows will possibly require re-activation at least (ever since Windows XP) if the drive is moved to another machine. Other than the operating system, the files are most likely accessible either way. So you could Install GNU/Linux on a machine, add that laptop drive as a second drive, and use GNU/Linux to copy the files over to something else before re-purposing it.

Laptop drives on average run for about 5 years, and the warranty is usually for 1 year, except for a very high-quality (above average) drive. This includes SSD drives, although traditional drives ("spin drives") often fail mechanically, and thus (if it's worth the price) can often have the data recovered after they fail.

They also frequently fail slowly, so they are often recoverable (at least a good percentage of the drive) even when the OS won't load, if you can connect them to a machine with a working OS (or boot one from USB.)

SSD drives are faster, and thus a good place to put the operating system, though they tend to fail very fast (all at once) and then the data is very hard or impossible to recover. It is good to do backups, but particularly of SSD drives.

"**Aren't SSD drives flash-based, like USB drives?**"
Yes, and often higher quality or capacity or performance.

"**But doesn't that mean...**"

Yes, pretty much.

Each type of connector on the main computer board corresponds to the "data bus" associated with that connector type. In the 1980s, there was an XT bus followed by a backwards-compatible (that is, it worked with older XT cards) ISA bus. XT was 8-bit and ISA was 16. The ISA connector was basically an XT connector with an extended portion, so you plugged the older cards into the XT portion, and if you

had an ISA card, it would have tabs for both portions of the "slot."

For a while, when the PCI bus came out, you could find boards that had some PCI slots and some ISA slots. PCI slots were smaller, had more connectors that were closer together, and generally represented progress in manufacturing and engineering. They were also 32-bit. Today, PCI Express slots are common, but you can still find boards with multiple connectors for different bus types.

Apart from older ISA slots, boards with PCI slots often had one or two "AGP" slots, which mostly only caught on for graphics cards. This is mentioned mostly as an example of Boards having more than one type of slot for more than one type of card. If you only have two types of card slot-- one larger type and one that is tiny-- then the larger size is probably for cards and (somewhere else on the board) the tiny slots are for ram modules.

But if your card fits the slot, it is likely compatible with the board. A possible exception is RAM-- with RAM modules, you need to be very specific. It's better if they match, and sometimes they have to be installed in a certain order. (You can read more about this in a more detailed repair manual-- online or at a library.)

It is possible for a card that is having trouble to cause problems elsewhere in the system. If seated improperly, it could prevent the system turning on or even damage the board or other components. The same if a board is shorted out. It could be a software issue instead of hardware-- if an improper software driver is installed or configured improperly, that device could be causing software problems that interfere with other parts of the system.

Isolating and fixing computer problems is something that takes learning and it improves with practical experience, but often the problem is simple and can be found with experimentation.

Good practices are taking notes, being honest with technicians (a lot of people are too shy to tell a technician what really happened, which only means the tech spends longer figuring it out for themselves and it still costs the same to fix, and per-hour charges took longer, so being honest is cheaper on average than being shy or proud) and handling everything with care. If something doesn't fit, whether inside our outside the computer, forcing it will probably only result in damage that sometimes can't be (economically) repaired.

If your computer has trouble, popular causes are: **1.** software configuration **2.** someone dropped it **3.** liquid spills / poor ventilation / dust **4.** power surges / unreliable or noisy power lines **5.** static discharge to internal components / improper handling **6.** something broke when it was forced or plugged in from a careless angle (people do that with headphones and USB all the time.) **7.** components age, even when turned off. They age faster when on. Sometimes, replacement is cheaper than repair. **Also:** don't buy touchscreen laptops.

Chapter 10: What's next?

This book (This version of the book) is in the public domain, and another version of the book could be created with the changes licensed CC BY 4.0 for the OER community. At that point, the book itself would be under the CC BY license (but most of it would still be in the public domain, as would the older versions of the book-- this version, at the very least.)

One thing that would help this book a lot is pictures. Then again, pictures can mislead as well as inform. But it would probably be a great idea to take this book and add CC BY licensed pictures to it, so that it would appeal to more people (and pictures can still help.)

Something else that could help this book is YOU. You could suggest other related topics for this book to cover. You could add them yourself-- or you could split this book into chapters (most of the work is already done there) or into individual topics and put them on a blog or website (it's public domain, so feel free to do so.) You could even print this book and sell it (be my guest.) Maybe you could make Youtube videos out of it.

But if you have read this book and would like to know more, let me know. Let someone know. And please feel free to share and modify this book. It belongs to no one; it belongs to everyone.

figosdev, Sept. 2018